RTA-TRACE

RTA-TCEL User Guide

Contact Details

ETAS Group

www.etasgroup.com

Germany

ETAS GmbH
Borsigstraße 14
70469 Stuttgart

Tel.:+49 (711) 8 96 61-102
Fax:+49 (711) 8 96 61-106

www.etas.de

USA

ETAS Inc.
3021 Miller Road
Ann Arbor, MI 48103

Tel.: +1 (888) ETAS INC
Fax: +1 (734) 997-94 49

www.etasinc.com

Japan

ETAS K.K.
Queen's Tower C-17F,
2-3-5, Minatomirai, Nishi-ku,
Yokohama, Kanagawa
220-6217 Japan

Tel.: +81 (45) 222-0900
Fax: +81 (45) 222-0956

www.etas.co.jp

France

ETAS S.A.S.
1, place des États-Unis
SILIC 307
94588 Rungis Cedex

Tel.: +33 (1) 56 70 00 50
Fax: +33 (1) 56 70 00 51

www.etas.fr

Korea

ETAS Korea Co. Ltd.
3F, Samseung Bldg. 61-1
Yangjae-dong, Seocho-gu
Seoul

Tel.: +82 (2) 57 47-016
Fax: +82 (2) 57 47-120

www.etas.co.kr

Great Britain

ETAS UK Ltd.
Studio 3, Waterside Court
Third Avenue, Centrum 100
Burton-upon-Trent
Staffordshire DE14 2WQ

Tel.: +44 (0) 1283 - 54 65 12
Fax: +44 (0) 1283 - 54 87 67

www.etas-uk.net

Copyright

Document TD00101-002

Contents

1 About this Manual

RTA-TRACE is a software logic analyzer for embedded systems. Coupled with a suitably enhanced application or operating system, it provides the embedded application developer with a unique set of services to assist in debugging and testing. Foremost amongst these is the ability to see exactly what is happening in a system at runtime with a production build of the application software.

This manual describes the RTA-TRACE add-on *RTA-TCEL* – a CAN-based ECU Link.

This manual does not explain how to instrument application code or build the target application with tracing enabled. For this you should consult both the *RTA-TRACE Getting Started Guide* and the *RTA-TRACE User Manual.*

More information about ECU Links can be found in the *RTA-TRACE ECU Link Guide* included with RTA-TRACE.

1.1 Who Should Read this Manual?

The *RTA-TCEL User Guide* is for the software engineer who wishes to use a CAN bus to transfer trace data from an application instrumented with RTA-TRACE.

It is assumed that the *RTA-TRACE Getting Started Guide* has been followed and an application has been built (and seen to work) using the RS232 ECU Link supplied with RTA-TRACE.

1.2 Document Conventions

Important: Notes that appear like this contain important information that you need to be aware of. Make sure that you read them carefully and that you follow any instructions that you are given.

Portability: Notes that appear like this describe things that you will need to know if you want to write code that will work on any target processor.

Program code, header file names, C type names, C functions and API call names appear in the `courier` typeface. When the name of an object is made available to the programmer the name also appears in the `courier` typeface, so, for example, a task named Task1 appears as a task handle called `Task1`.

`Courier oblique` is used for placeholders where the user should substitute relevant text, e.g. `RTser`*`xxx`* refers to both `RTserbyt` and `RTserblk` libraries.

When interaction with GUI elements is described, the elements' captions are shown in **bold**. Navigation of a hierarchy, such as a menu structure, is shown

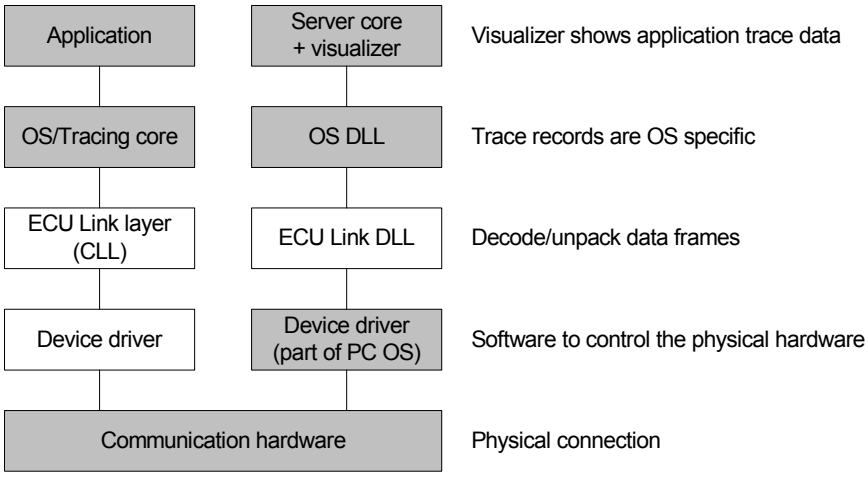
by separating the levels with a chevron, e.g. "Choose the **Edit > Select All** Menu item." Or "Choose **Edit > Select All**".

2 What is RTA-TCEL?

RTA-TRACE requires an *ECU Link* to transfer data from the target to the RTA-TRACE Server. RTA-TCEL is a CAN-based ECU Link.

The *RTA-TRACE ECU Link Guide* provides some more background information about ECU Links.

The general layering of the communications architecture is illustrated below:



In this reference model, the target code consists of two elements and the server code consists of one.

RTA-TCEL provides a means of transferring RTA-TRACE data from an instrumented target using the CAN bus. CAN allows higher data rates to be used (up to 1 Mbit) and is therefore better suited for tracing applications which generate larger amounts of trace data. CAN is also a shared bus, so it is possible to communicate with more than one application sharing the same bus.

2.1 RTA-TCEL package contents

For RTA-TCEL, the 'Device Driver' block in the diagram above is actually composed of two parts – an ELL adaptation layer and a CAN device driver. The CAN device driver is intended to offer a simple interface to the CAN hardware, and may be replaced by another device-driver if necessary.

Note: Target code is supplied for use with the Diab/MPC56x version of RTA-OSEK . Other OS/targets will require some code modification. The adaptation layer is written to be target-independent, although some changes may be necessary.

The RTA-TCEL package consists of the following:

- Software CD containing:
    - RTA-TRACE Server TCEL DLL;
    - Target device-driver example code;
    - Target ELL adaptation-layer example code.

Each of these will be described in further detail later.

3 Installing RTA-TCEL

There are two aspects to the installation process – host PC hardware/software, and target software.

3.1 Host PC

The RTA-TCEL software is provided on a CD.

1. Insert the RTA-TCEL CD into the drive;
2. If the installer program does not start automatically, locate and run the '`setup.exe`' file on the CD; a splash screen will appear.
3. Clicking on 'Install' will start the RTA-TCEL Windows installation wizard. You will be asked for the location of the license file for RTA-TCEL, this will have been supplied separately. When installation is complete, click the 'Finish' button.

3.2 Target Software

The Software CD contains a directory called `targets\diab56x`, the following files are contained there:

`CAN_dd.c`	This file contains an implementation of a CAN device driver. This is a reference implementation solely for support of the CAN ECU Link. This file will need to be re-written for your particular target processor.
`CAN_dd.h`	This file contains the function prototypes for the example CAN device driver. This file should be included into any other source file which uses the example CAN device driver.
`TCEL_s.c` `TCEL_m.c`	These files each contains an adaptation layer between the ECU Link library included with RTA-TRACE and the above CAN device driver. `TCEL_s.c` uses a single CAN frame for trace-data, `TCEL_m.c` uses multiple CAN frames.
`config_s.h` `config_m.h`	Configuration header files for the single- and multiple-frame adaptation layers.
`M563HEADER.zip`	A support file from Motorola describing the register layouts for the MPC563 microcontroller. This file is only necessary if this is the processor being used. For other Motorola 56x devices, similar header files are available from Motorola.

These files should be copied to your application build location as appropriate. (Note that the files may be marked as read-only if copied straight from the CD – they will need to be made writable if editing them)

4 Usage

RTA-TCEL consists of software running on both target and PC.

4.1 Target software

The RTA-TCEL target software files (listed in Section 3.2) consist of an example CAN device driver and adaptation-layers which will integrate with the serial ECU Link libraries included with RTA-TRACE component of the OS (See *RTA-TRACE ECU Link Guide* for more details of the required API; alternatively the `RTRS232.c` driver file supplied with RTA-TRACE or you operating-system is an example of the required API). The example code has been written for RTA-OSEK running on the MPC56x using the Diab compiler suite. The code will require porting for other targets. LiveDevices can offer assistance in this aspect.

The basic steps for integrating the target software into your application are the same as for the Serial ECU Link (See *RTA-TRACE ECU Link Guide* for full details of the required API).

4.1.1 Frame configuration

The configuration file used by RTA-TCEL contains information about:

- the CAN frame-identifiers used;
- the number of CAN frames;
- whether standard or extended identifiers are used.

Note: Care should be taken when allocating frame identifiers when integrating RTA-TCEL with an existing CAN-based application, since an incorrect identifier choice may compromise the performance of the application being measured.

A configuration file looks something like this:

```
#define LD_STANDARD_CAN_IDS  (1)
#define LD_NUM_CAN_FRAMES    (1)
#define LD_CAN_FRAMES        0x123
```

As can be seen, this looks like a standard C header file – this is intentional: it allows consistency to be maintained between Server and target configurations. An example of this type of file can be found in `config_s.h` in the target directory.

In the case where multiple CAN frames are being used to transfer data, LD_CAN_FRAMES becomes a comma-separated list of identifiers as follows:

```
#define LD_STANDARD_CAN_IDS  (1)
#define LD_NUM_CAN_FRAMES    (3)
#define LD_CAN_FRAMES        0x123,0x124,0x125
```

This allows `LD_CAN_FRAMES` to be used to initialize an array of identifiers in the target C code (see `TCEL_m.c` and `config_m.h` for an example of this).

In the configuration file, `LD_STANDARD_CAN_IDS` can be replaced by `LD_EXTENDED_CAN_IDS` if extended identifiers are required.

Configuration files can be written using any text editor.

The supplied example code is described in Section 5.

Note: if the configuration files are not consistent between the target software and RTA-TRACE Server, data may be corrupted, or not be seen.

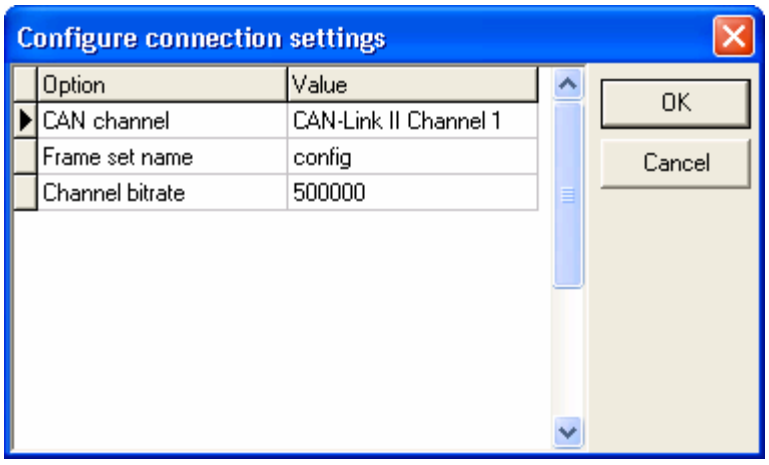
4.2 Configuring the RTA-TRACE Server

ECU Link DLLs appear as '<*OS*>-<*ECU Link*>' choices in the RTA-TRACE Client connection setup dialogue – the CAN ECU Link appears as '*OS*-TCEL'.

The ECU Link can be configured either at the Server or the Client:

- Server Method: Right-click the RTA-TRACE Server icon in the System Tray. Click the **Properties** menu item and choose the relevant *operatingsystem*-TCEL menu item. Changes to the settings take effect when you click **OK** to close the dialog box.
- Client Method: Choose **File** > **Configure Connection…** . A dialog appears showing the configurable options. Changes take effect when the **OK** button is clicked.

The configuration options in both cases allow the CAN interface, the CAN frame-set, and the bus bitrate to be set.



Configuration files must have a `.h` extension and be placed in the `<RTA-TRACE installation directory>\TCEL_config\` directory. The directory is re-read every time the configure-dialog is opened.

(For example, the above dialog box is the result of there being a configuration file called `config.h` in the ...`\TCEL_config` directory)

5 Target Examples

The supplied software comprises a device driver (`CAN_dd.c`) with two adaptation layer variations, these are:

- Single CAN frame used for trace data, block-serial ELL (`TCEL_s.c`);
- Multiple CAN frames used for trace data, block-serial ELL (`TCEL_m.c`).

The API implemented by the `TCEL_` files is specified in the *RTA-OSEK ECU-Link Guide*.

Using a single CAN frame to transmit trace data is easy to set up, and has minimal impact on any existing CAN frame-ID allocation scheme, but will not provide a particularly high data-rate solution since there will be a response-gap between the CAN controller becoming available for transmission, and the ECU Link being able to supply data.

It is possible to attain a higher data-rate by using multiple slots in the CAN controller in conjunction with multiple CAN frame IDs. Since the CAN protocol has a well-defined priority scheme, it is possible to load trace-data into multiple CAN frames (highest priority frame ID first) and let the CAN controller transmit them back-to-back.

For the examples, the same device-driver has been used. To port the examples to a different target, the device driver is the only element which will have to be re-written if the same device-driver API is implemented.

5.1 Basic steps

For this walkthrough, we will assume that you have an application with tracing enabled and are using the RS232 example drivers supplied with RTA-TRACE. This stage can be reached by following the steps described in the *RTA-TRACE Getting Started Guide*.

For simplicity, we are also assuming that this process is being carried out using the default configuration (RTA-OSEK/Diab MPC56x) built for a MPC563.

The following example will result in a single CAN frame identifier being used to transfer data.

1. Copy the following files from the TCEL target software directory to the application directory currently containing the `RTrs232.c` file:
   - `CAN_dd.c`
   - `CAN_dd.h`
   - `TCEL_s.c`
   - `TCEL_m.c`
   - `config_s.h`
   - `config_m.h`
   - `M563HEADER.zip`
2. Modify the configuration structure in `osTraceCommInitTarget()` (In `TCEL_s.c`) appropriately. The default configuration (contained in the `cfg_block` structure) assumes a 40MHz system clock and a

required bit rate of 500kbit/s. The remaining parameters of the configuration structure refer to CAN parameters that are fully described in the *CAN 2.0B specification document* from Robert Bosch.

3. Change your build process, removing `RTrs232.c`, and adding compilation of `CAN_dd.c` and `TCEL_s.c`;
4. Change the ELL library in your build script from `RTAserbyt` to `RTAserblk`. (if using the OS Instrumenting Kit, build `RTutd.c` in block-mode by ensuring `BLOCK_MODE` is defined at compilation).
5. If the `M563HEADER` file has not already been installed, unpack `M563HEADER.zip` so that the subdirectory `M563HEADER` exists within the application directory (if you require the header files elsewhere, then `can_dd.h` will need to be updated to reflect the location of `M563HEADER`).
6. Build the application.
7. The application can now be run. The supplied configuration file (`config_s.h`) will generate CAN frames of up to 8 bytes with the standard identifier `0x123`.
8. Copy the `config_*.h` files into the `<RTA-TRACE installation directory>\TCEL_config\` directory, and start RTA-TRACE, using the <*OS*>-TCEL configuration.
9. Ensure that the connection is configured correctly (either by selecting **Configure Connection...** from the **File** menu, or by using the **Properties > <*OS*>-TCEL** popup menu from the RTA-TRACE Server taskbar icon) to the correct interface, the correct bit rate, and the correct frame-set (*config_s*).
10. Trace data should now be visible. If there is no data visible, check the diagnostic windows for the Server and <*OS*>-TCEL link.

5.1.1 Adaptation layer details (TCEL_s)

The single-frame adaptation layer (`TCEL_s.c`) allocates an 8 byte buffer for the CAN frame data which is filled by the ELL. When the ELL fills the buffer (or runs out of data) it makes a call to `osTraceCommTxBlock()` with the number of bytes to send. The function then simply calls the device driver function `canTxMsg()` with the parameters set to send a frame with the ID 0x123 (using the macro defined in the configuration file) from the first slot in the CAN controller.

The implementation of `osTraceCommTxReady()` calls `canStatus()` if a CAN frame recently been sent, returning non-zero if the slot status is `CAN_SLOTEMPTY`. If a CAN frame has not been sent recently, non-zero is returned.

5.2 Customization

5.2.1 Using multiple CAN frames to carry trace data

The example described above uses a single CAN frame to transfer trace data. It is possible to achieve a greater data throughput by using multiple CAN frames.

In the above example, the multi-frame adaptation layer can be used instead of the single frame version by changing `TCEL_s.c` to `TCEL_m.c`. This process will also change the configuration file used (to `config_m.h`).

Note: The supplied `TCEL_m.c` expects `LD_CAN_FRAMES` to be defined with the highest priority (lowest number) CAN ID first. Failure to maintain this order will result in trace data being corrupted since the data frames will be transmitted out-of-order.

5.2.2 Adaptation layer details (TCEL_m)

The multi-frame adaptation layer (`TCEL_s.c`) allocates a buffer for the CAN frame data which is filled by the ELL. The size of the buffer is given by the number of CAN frames in use (taken from the configuration file) multiplied by the size of each CAN frame (specified within `TCEL_m.c`). When the ELL fills the buffer (or runs out of data) it makes a call to `osTraceCommTxBlock()` with the number of bytes to send. The function then calls the device driver function `canTxMsg()` a number of times, with the parameters set to send a set of frames with the IDs defined in the configuration file. Multiple slots in the CAN controller are used (one slot per CAN frame) which means that once the frames have been written, the CAN controller itself is responsible for sending the frames onto the bus as quickly as possible.

The implementation of `osTraceCommTxReady()` calls `canStatus()` if a CAN frame recently been sent, returning non-zero if the slot status is `CAN_SLOTEMPTY`. If a CAN frame has not been sent recently, non-zero is returned.

5.2.3 Modifying the Device Driver

For targets where the supplied CAN device driver is not appropriate, it will be necessary to re-write it. See Section 6 for a description of the device driver functionality. Maintaining the same device-driver API will mean that the TCEL_ layers will not need to change.

5.2.4 Using a third-party CAN Device driver

The simple device-driver supplied is intended to be a generic implementation. Conversion of either `TCEL_s.c` or `TCEL_m.c` to use a different device-driver

should be straightforward. Refer to Section 6 for further details of the supplied device driver.

6 Device Driver reference

The device driver supplied presents the following simple API to the adaptation layer.

6.1 Required Types

Certain types are used by the CAN device driver which must be supplied by the OS or user code. The default software configuration (for RTA-OSEK) uses the RTA-OSEK include file `ostarget.h` to define these types.

The required types are as follows:

`UInt32Type` An unsigned 32 bit type

`ByteType` An unsigned 8 bit type

6.2 Defined types

6.2.1 canStatusType

The device driver calls each return a status code. The code is defined as follows:

```
/* Status return values for CAN Device driver calls
 */
typedef enum {
    CAN_OK = 0,
    CAN_UNINIT,
    CAN_SLOTBUSY,
    CAN_SLOTEMPTY,
    CAN_SLOTNUM,
    CAN_ERROR = 0xFF
} canStatusType;
```

The meaning of each return code is described with each API call.

6.2.2 canConfigType

To configure the CAN interface, the following structure exists which encapsulates the configuration data necessary for a CAN controller. The names of the variables are as specified in the CAN 2.0B specification document.

```
/* Configuration data block for
 * CAN Device initialization
 */
typedef struct {
    UInt32Type sysclock;
    UInt32Type bitrate;
    ByteType   propseg;
    ByteType   pseg1;
    ByteType   pseg2;
    ByteType   rjw;
    ByteType   tx_slots;
} canConfigType;
```

Some knowledge of the CAN controller in use will be necessary to ensure that a compatible set of parameters are specified. For the supplied system, bit rates of 1Mbit, 500kbit, 250kbit, 125kbit, and 62.5kbit are achievable (simple factors of two from the default setting) simply by changing the bitrate element. Other bit rates or using a different system clock rate may affect the rates available.

Please refer to the manual for your CAN controller if specifying custom bitrates.

6.3 API reference

In the following reference, CAN controller slots are numbered from zero.

6.3.1 canInit

Prototype

```
canStatusType canInit(
                    canConfigType *config)
```

Description

This function is responsible for initializing the CAN device driver.

It is required to initialize the CAN controller to a given bit rate as well as set up a specified number of controller slots for transmission.

For the supplied device driver, the initialization code

calculates the number of *time-quanta* per bit, and based on that value and the requested bit rate, derives a prescaler value in order to generate a suitable clock from the given system clock frequency. The function then marks the requested number of controller slots suitable for use as transmit slots.

Return `CAN_OK` is returned if initialization was successful, `CAN_ERROR` otherwise.

6.3.2 canTxMsg

Prototype

```
canStatusType canTxMsg(
                    ByteType slotNum,
                    UInt32Type msgId,
                    ByteType *data,
                    ByteType dataLength)
```

Description This function transmits a CAN message.

`slotNum` contains the slot used for transmission of this frame. The number of slots available will vary between CAN controllers, CAN_ERROR is returned if the slot does not exist.

`msgId` contains the full frame Identifier including RTR, SRR, and IDE flags (where applicable). This means that a standard identifier is 12 bits long (11 bit ID + RTR bit), and an extended identifier is 32 bits long (29 bit ID + RTR, SRR, IDE bits). Standard identifiers are contained in the least-significant 12 bits. Helper macros are supplied in `CAN_dd.h` which pack a given frame ID into a form suitable for use in this argument.

`data` points to the data block which makes up the frame body. This will contain between zero and eight bytes.

`dataLength` is the length of the data block pointed to by data. If `dataLength` is greater than 8 bytes, only the first eight bytes are transmitted.

Return `CAN_OK` when the frame has been successfully transferred to the controller;

`CAN_SLOTNUM` if the specified is not valid.

6.3.3 canStatus

Prototype

```
canStatusType canStatus(
                    ByteType SlotNum)
```

Description This call returns the status of the transmit CAN controller slot specified with `SlotNum`.

Return

`CAN_SLOTNUM` if the specified slot is not valid;

`CAN_UNINIT` if the controller has not yet been initialized;

`CAN_SLOTBUSY` if the slot is currently in use;

`CAN_SLOTEMPTY` if the slot is empty.

Support

For product support, please contact your local ETAS representative.

Office locations and contact details can be found on the ETAS Group website www.etasgroup.com.